Department of Computing
電子計算學系

COMP2021 Object-Oriented Programming

(Fall 2022)

Project Report

Group 6

LIU Chenxi 21096794d

Zhang Wanyu 21094724d

ZHANG Wenxuan 21099467d

DU Haoxun 21097883d

1 Introduction

This document describes the design and implementation of an interpreter for Simple programs by group#. The project is part of the course COMP2021 Object-Oriented Programming at PolyU.

There are three parts in this report, which respectfully are the introduction (part 1), a command-line-based interpreter for simple programs (part 2) with two detailed small sections, and the user manual (part 3). In each of them, we will illustrate the project in different aspects to let readers understand our design and program profoundly and thoroughly. Comprehensively, part 1 is the introduction of the overall of this project, which contains the brief function and target intro with the default development environment. Part 2 is the specific description of the coding part with the design of the entire program and the achievements of every single requirement. Part 3 is the user manual that how the interpreter works from a user's perspective, listing all the commands the system recognizes in this section while using screenshots for each command to see the output for various inputs.

In our daily, no matter how advanced programmers we are, the use of Command Prompt (also named a terminal in Mac system) cannot be avoidable. An executable command-line interface that is intended to prompt response is known as a command prompt. By entering commands into a Command Prompt, you may launch applications, change Windows settings, and access files.

The objective of this project is to provide a command-line-based interpreter for Simple programming language applications. Similar to Java, Simple is a programming language that only supports a small number of data types, operators, and statement kinds. Users may write and understand Simple programs using the interpreter. The development environment is based on the JAVA17 with all the used libraries are built-in libraries in the JAVA17.

2 A Command-Line-Based Interpreter for Simple Programs

In this section, we describe first the overall design and then the implementation details of the interpreter.

2.1 Design

We divide the whole program into several main parts, which are the main functions, operation class, the HashMap setting function, and the checking function with a few Enum for the rename of the types.

For the main function:

Read-in loop, which allows user to input commands for multiple times unless they quit themselves.

```
while (true){
    System.out.print(">");
    Scanner scanner = new Scanner(System.in);
    String com= scanner.nextLine();
    String[] inputlist = com.split( regex: " ");
```

In this block, we achieve the read-in function. All the input will be considered as the String type first, and we immediately change it into a String array by "split" (built-in), which is pretty convenient for us in the following work as we can position each certain expression with matched index in the array.

We intentionally use switch to catch the operation of the input commands and will find the matched class to do the further execution.

```
switch (inputlist[0]) {
case "vardef":
    setHashMap.getStatementMap().put(inputlist[1], new varDef(com));
    break;
case "binexpr":
    setHashMap.getExpressionMap().put(inputlist[1], new binaryExp(com));
    break;
case "unexpr":
    setHashMap.getExpressionMap().put(inputlist[1], new unaryExp(com));
    break;
case "assign":
    setHashMap.getStatementMap().put(inputlist[1], new Assign(com));
    break;
case "print":
    setHashMap.getStatementMap().put(inputlist[1], new Print(com));
    break;
case "skip":
    setHashMap.getStatementMap().put(inputlist[1], new Skip(com));
    break;
```

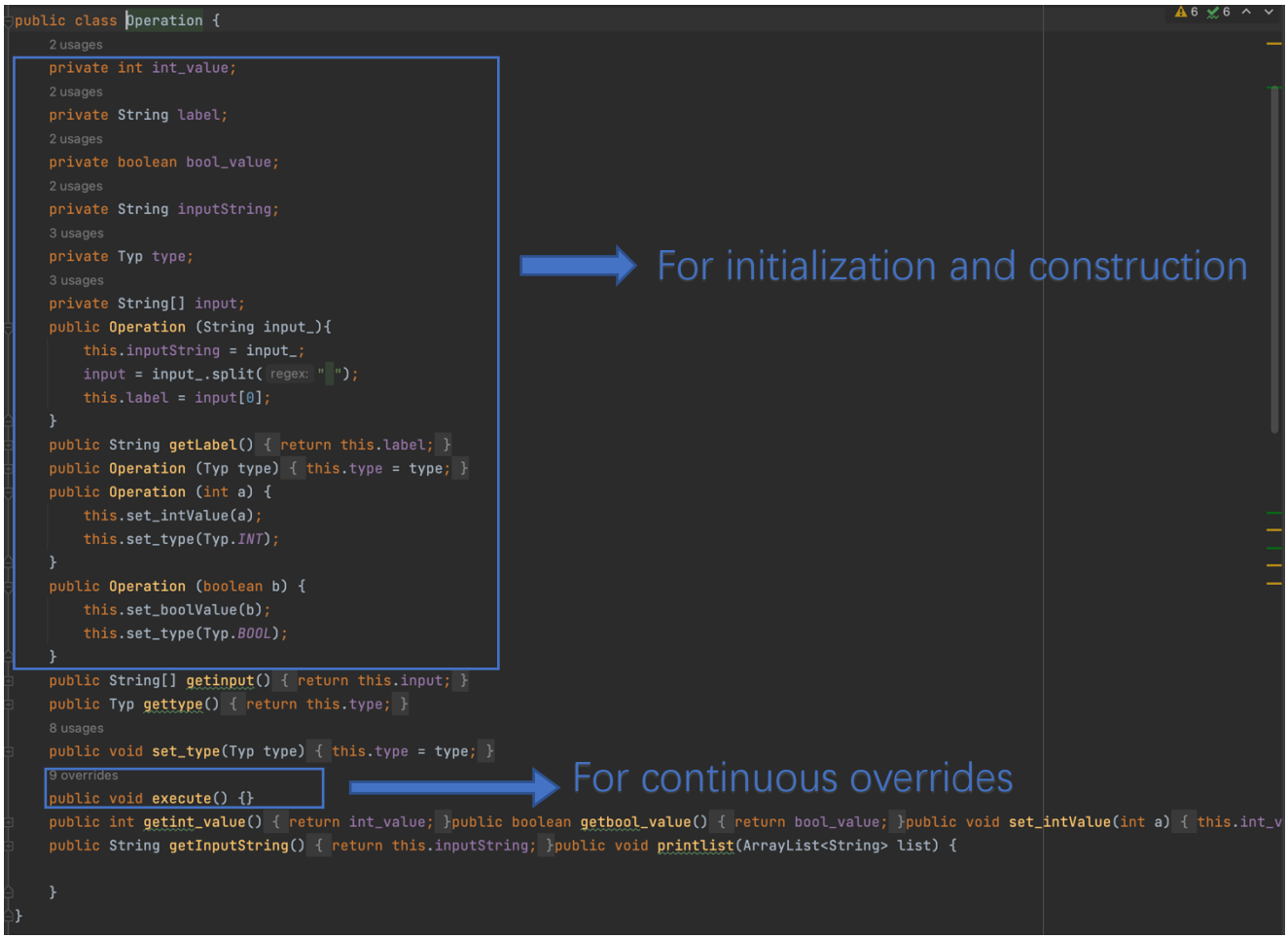
This block is the case-checking part inside the switch. It will find the coordinated function class and transfer the input into it to execute.

Ps: the function name will be placed in the first of the input array by default.

Etc. As the switch cases still have several, so will not be shown as it will take a lot of space.

For the operation class:

This class is the superclass of the majority of the following function classes which will be seen in the required part. We consciously use this method to not only bring out the best of the java language (OOP's characteristic) but also minimize the amount of duplicate code in an application, having a better organization of code with the advantage that code can be more flexible to change. You will find that the function classes mostly extend this class and then do some changes or overrides based on this fundamental class.




```
public class Operation {
    2 usages
    private int int_value;
    2 usages
    private String label;
    2 usages
    private boolean bool_value;
    2 usages
    private String inputString;
    3 usages
    private Typ type;
    3 usages
    private String[] input;
    public Operation (String input_){
        this.inputString = input_;
        input = input_.split( regex: " ");
        this.label = input[0];
    }
    public String getLabel() { return this.label; }
    public Operation (Typ type) { this.type = type; }
    public Operation (int a) {
        this.set_intValue(a);
        this.set_type(Typ.INT);
    }
    public Operation (boolean b) {
        this.set_boolValue(b);
        this.set_type(Typ.BOOL);
    }
    public String[] getinput() { return this.input; }
    public Typ gettype() { return this.type; }
    8 usages
    public void set_type(Typ type) { this.type = type; }
    9 overrides
    public void execute() {}
    public int getint_value() { return int_value; }public boolean getbool_value() { return bool_value; }public void set_intValue(int a) { this.int_v
    public String getInputString() { return this.inputString; }public void printlist(ArrayList<String> list) {

    }
}
```

For the first block (the upper one), it is the initialization part of the whole class, and we set a few constructors to make this class can construct different variables based on the actual use of this superclass.

To emphasize, in the second block (the lower one), it is the execute method that will be overridden in the following inheritance based on the real function of the certain class, to realize the corresponding function.

For the HashMap setting function:

Java's Map interface is implemented using a hash table in Java HashMap. As you may already know, a map is a grouping of key-value pairs. Keys and values are mapped. The reason why we choose the HashMap data structure to be the main container of our project is that the Java HashMap is thread-insecure, it is necessary to explicitly

synchronize any concurrent HashMap updates. What's more, Java HashMap accepts null keys and null values with an unordered collection as a hash map. The elements' specific sequence is not guaranteed by it.

```
import java.util.HashMap;

public class setHashMap {
    1 usage
    public static HashMap<String, Operation> statements = new HashMap<>();
    1 usage
    public static HashMap<String, Variable> variables = new HashMap<>();
    1 usage
    public static HashMap<String, Operation> expressions = new HashMap<>();
    1 usage
    public static HashMap<String, Program> programs = new HashMap<>();
    32 usages
    public static HashMap getStatementMap() { return statements; }
    5 usages
    public static HashMap getProgramMap() { return programs; }
    16 usages
    public static HashMap getExpressionMap() { return expressions; }
    //not used
    public static HashMap getVariableMap() { return variables; }


}
```

We have four hash maps, and their roles are:

For the statement HashMap, it stores the command statements with its key.

For the variables, HashMap stores the variable name and its value where the keys stand for the variable name while the value is supposed to be the value of the variable.

The expression HashMap stores the command expression with its key.

For the programs HashMap, it is set for the following functions which are related to the program list and load, we store the program keywords in this HashMap for checking and reading.

Thus, we add four methods to get the HashMap for further use and the check, you can get a certain map by using its matched get method.

For the checking function:

As there are two variable types in the project, before using or checking the value, we have to know the type of the variable. To achieve that, we write a checking class to check whether the variable is Boolean or integer.

```java
public static boolean isNumeric(String strNum) {
    if (strNum == null) {
        return false;
    }
    try {
        int d = Integer.parseInt(strNum);
    } catch (NumberFormatException nfe) {
        return false;
    }
    return true;
}
1 usage
public static boolean isBoolean(String strbool) {
    if (strbool.equals("true")||strbool.equals("false")) {
        return true;
    }return false;
}
```

First, we have “isNumeric” and “isBoolean” these two methods examine the type of the input variable, with a Boolean return.

Although these two methods seem simple, they are the significant components of the following function methods as we all have to use them to judge the type of the variable.

“Getvalue” is the method that returns the value of the input in various conditions.

First judge: whether the input is the expression

Second judge: whether the input is the variable

Third judge: whether is integer or Boolean if it is not initialized

If the input is expression, judge whether is binary expression or unary expression

If the variable has been created, check the type and return

```java
public static Operation getvalue(String expName) {
    Operation exp = null;
    if (setHashMap.getExpressionMap().containsKey(expName)) {
        if (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("binexpr")) {
            ((binaryExp) setHashMap.getExpressionMap().get(expName)).execute();
            System.out.println(((binaryExp) setHashMap.getExpressionMap().get(expName)).getbool_value()+"1");
            exp = (Operation) setHashMap.getExpressionMap().get(expName);
            System.out.println(exp.getbool_value()+"2");
        }
        else if  (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("unexpr")) {
            ((unaryExp) setHashMap.getExpressionMap().get(expName)).execute();
            exp = (Operation) setHashMap.getExpressionMap().get(expName);
        }
    }else if (setHashMap.getVariableMap().containsKey(expName)) {
        if (((Variable) setHashMap.getVariableMap().get(expName)).getvalue() instanceof Integer) {
            System.out.println();
            exp = new Operation(((Variable<Integer>) setHashMap.getVariableMap().get(expName)).getvalue());
            System.out.println(this.exp.getint_value());
        }else if (((Variable) setHashMap.getVariableMap().get(expName)).getvalue() instanceof Boolean) {
            exp = new Operation(((Variable<Boolean>) setHashMap.getVariableMap().get(expName)).getvalue());
        }
    }else if (Checking.isNumeric(expName)) {
        exp = new Operation(Integer.parseInt(expName));
    }else if (Checking.isBoolean(expName)) {
        exp = new Operation(Boolean.parseBoolean(expName));
    }
    return exp;
}
```

This class is created to view the execute statement of different function classes. It will find the match class and let it execute.

From the statement HashMap, get the label of the input

```
public static void executeStatement(String name) {
    switch (((Operation) setHashMap.getStatementMap().get(name)).getLabel()) {
    case "vardef":
        ((varDef) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "assign":
        ((Assign) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "print":
        ((Print) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "skip":
        ((Skip) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "block":
        ((Block) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "if":
        ((IfElse) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "while":
        ((While) setHashMap.getStatementMap().get(name)).execute();
}
}
```

With the switch structure, we can find the matching case, and run the inside execute of the class.

These two methods are designed for the following list program function.

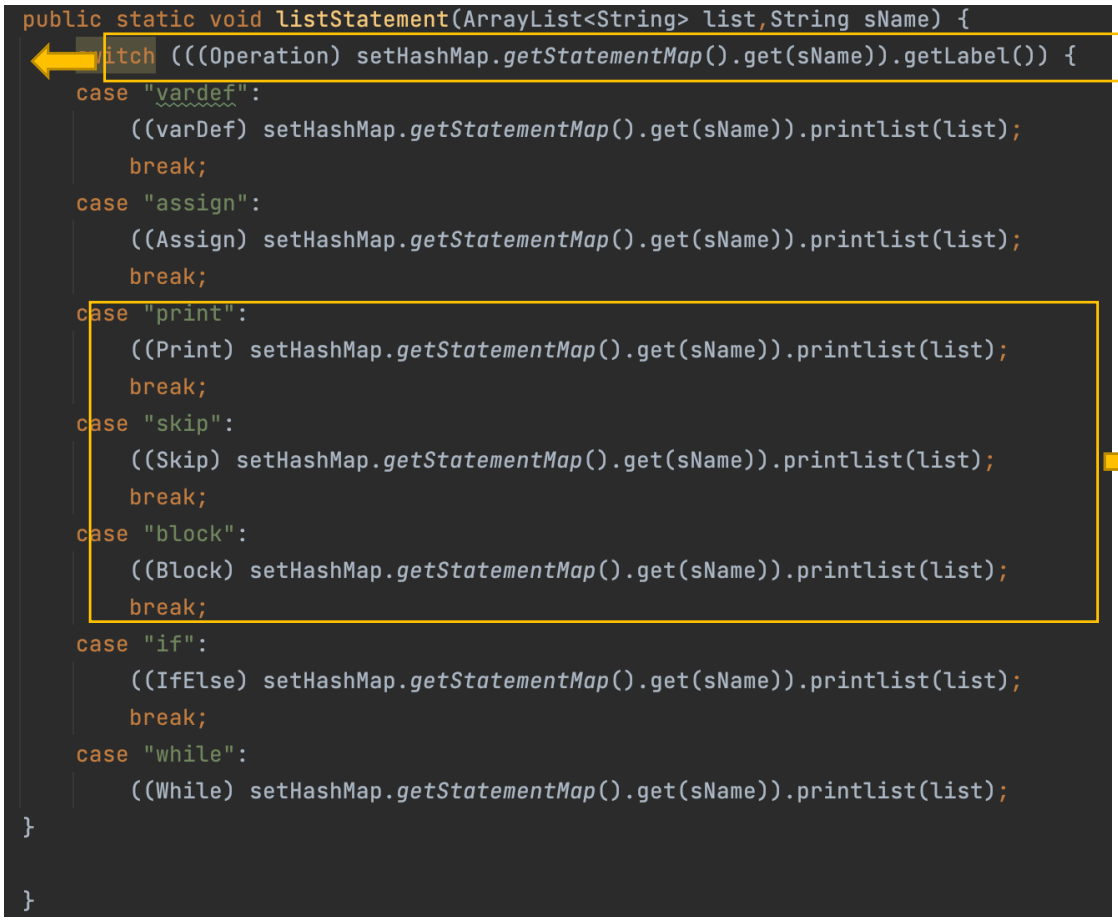
Check whether the expression exists

```
public static void listExpression(ArrayList<String> list,String expName) {
    if (setHashMap.getExpressionMap().containsKey(expName)) {
        if (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("binexpr")) {
            ((binaryExp) setHashMap.getExpressionMap().get(expName)).printlist(list);
        }
        else if  (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("unexpr")) {
            ((unaryExp) setHashMap.getExpressionMap().get(expName)).printlist(list);
        }
    }

}
```

Judge whether is binary expressio n or unary

If the input is not expression, it will be checked in the statement HashMap

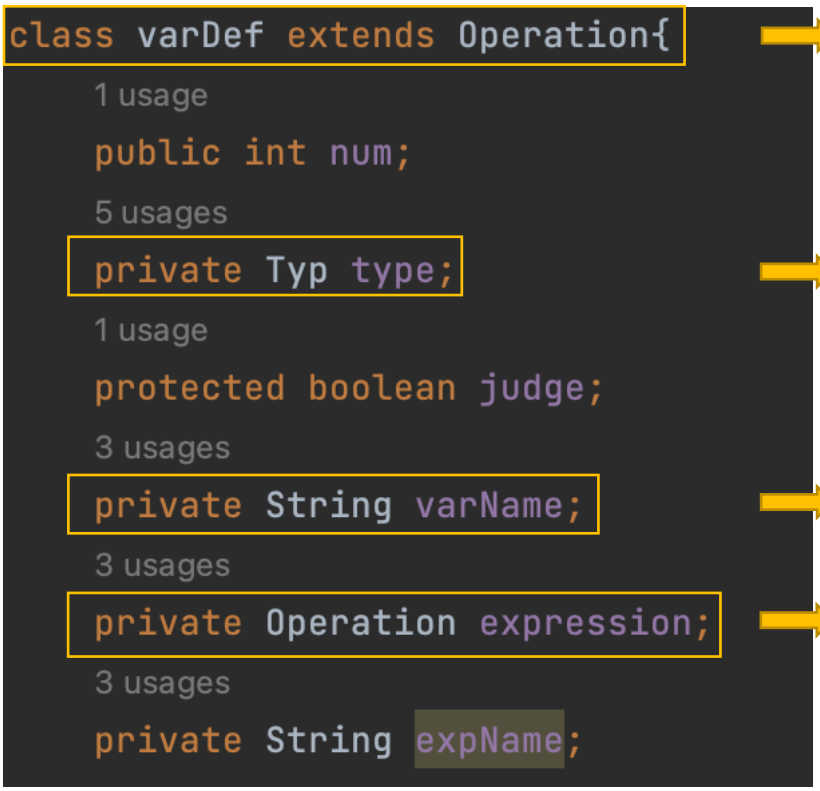


2.2 Requirements

[REQ1]

1) The requirement is implemented.

2) Implementation details.

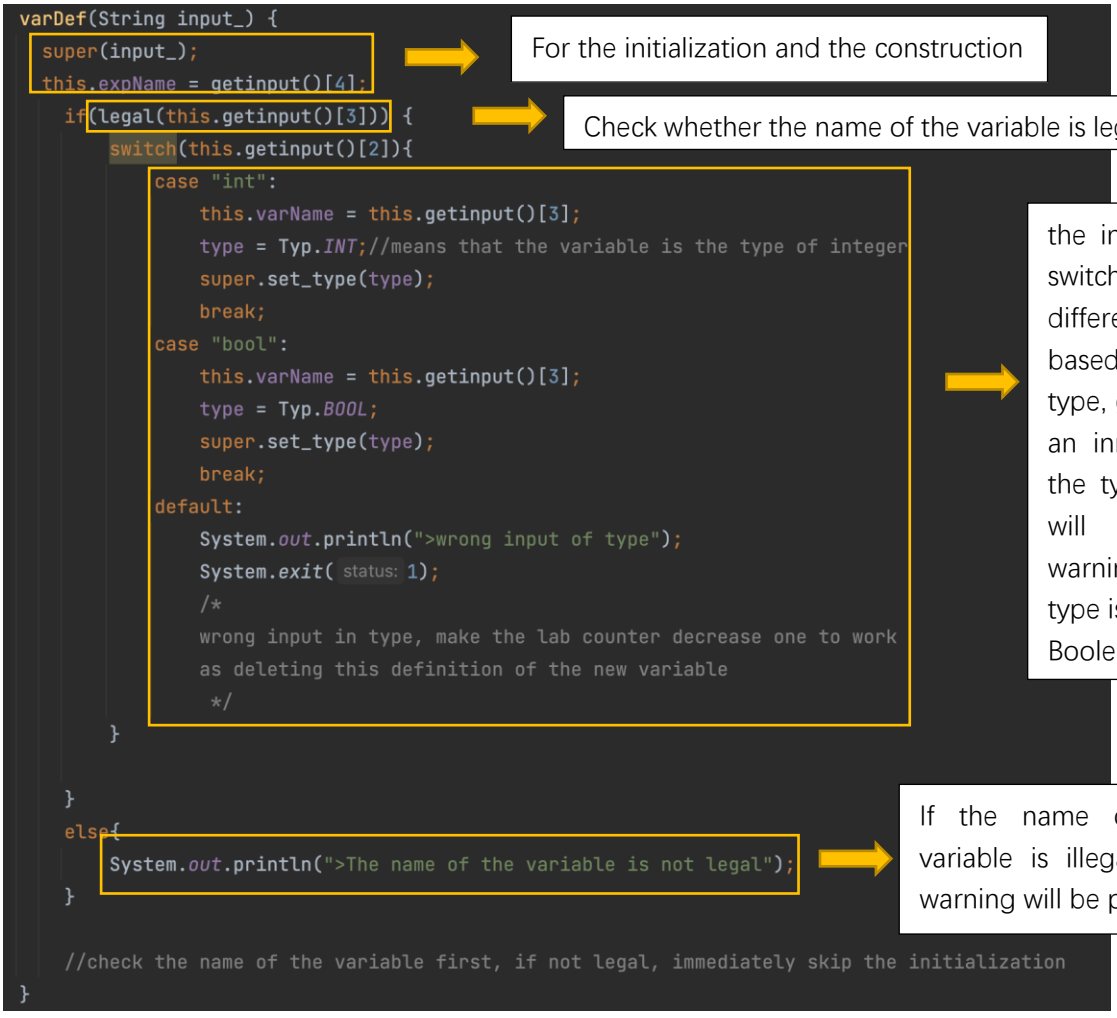


Inherit the Operation class

This type is a Enum, which is used as a sign to distinguish the int and Boolean type

To store the variable

To store the whole expression



For the initialization and the construction

Check whether the name of the variable is legal

the input will be switched into different case based on their type, giving them an inner sign of the type. Also, it will print out warning if the type is not int nor Boolean.

If the name of the variable is illegal, this warning will be printed

```java
public int get_IntValue() { return num; }

1 usage
public boolean get_BoolValue() { return this.judge; }

2 usages
public String get_VarName() { return this.varName; }
```

The simple method for getting the value of the variable. After calling this method, we will get the real value which is stored inside for two different types(which means for different types, we will use different methods to get value).

```java
private boolean isintchar (char a){
    if((a>=90&&a<=122)||(a>=65&&a<=90)||(a>=48&&a<=57)) return true;
    else return false;
}

//judge the name of the variable whether is legal or not
1 usage
private boolean legal(String name){
    if(name.length() > 8) return false; //length
    if(name.charAt(0)>= 48 && name.charAt(0) <= 57) return false;//not start with digit
    for(int i = 0; i < name.length();i++){
        if(!isintchar(name.charAt(i))) return false;
    }//only have letters and digits
    return true;
}
```

To guarantee all the characters should be char or number.

This method is created to check the legality of the name of the variable

```java
public void printlist(ArrayList<String> list) {
    Checking.listExpression(list, this.expName);

    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }
}
```

This “printList” method is created to work for the following function “program list”.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *lab* is valid
3. Whether the *typ* is int or bool
4. If the type is int, whether it follows int type *expRef* (is int string and in the range -99999~99999 or the expression name is valid)
5. If the type is bool, whether it follows bool type *expRef* (is bool string or the expression name is valid)
6. Whether the *varName* is valid
7. Whether the statement with the same *lab* is already stored

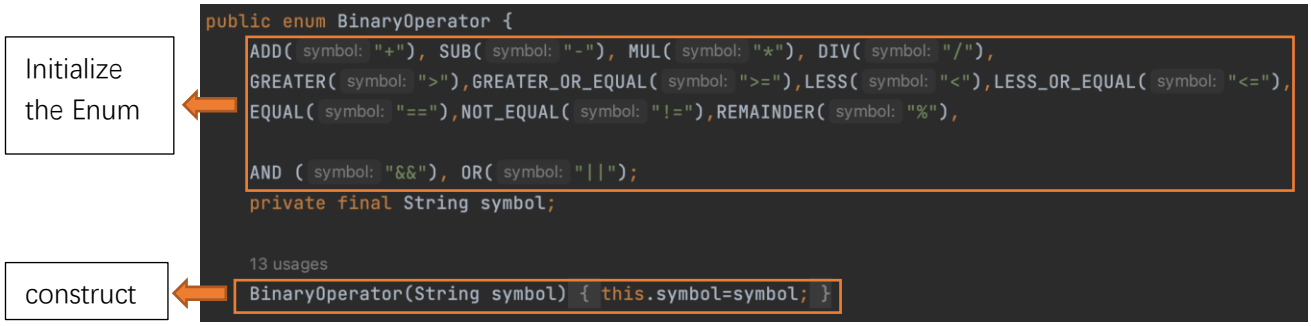
For execute error handling, it has the following checking:

1. Whether the *varName* has already been defined
2. Whether the type of *expRef* is corresponding to the *typ* of the variable. Because if the expRef is a expression or a variable, we can not handling the type in the user input format handling.

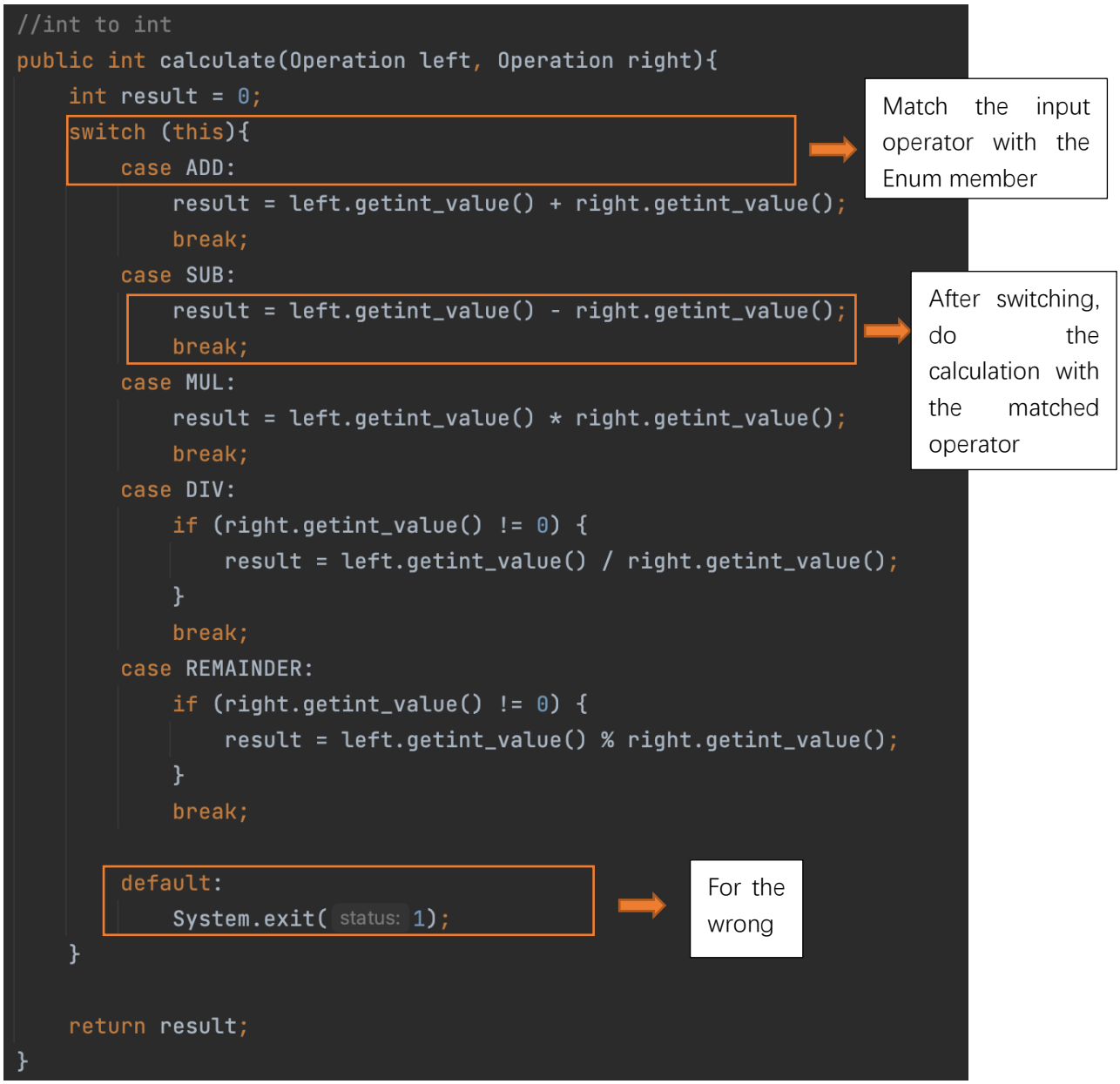
[REQ2]

1) The requirement is implemented.

2) Implementation details.



First, we create this Enum to store all the binary operators which will be covered in the input commands, and, we add a constructor to make it able to connect the input operator with its match Enum member.



This method is used to do the calculation between the integer and the integer, the switch function will find the match operator and put it between two variables to do the calculation.

Switch function for matching

The inside calculation of each small case

```
//boolean to boolean
public boolean bool_check(Operation left, Operation right){
    boolean bool_result = true;
    switch (this){
        case AND:
            bool_result = left.getbool_value() && right.getbool_value();
            break;
        case OR:
            bool_result = left.getbool_value() || right.getbool_value();
            break;
        case EQUAL:
            bool_result = left.getbool_value() == right.getbool_value();
            break;
        case NOT_EQUAL:
            bool_result = left.getbool_value() != right.getbool_value();
            break;
    }

    return bool_result;
}
```

This method is made to do the calculation of the two Boolean variables, which works the same way as the above calculate method.

Switch function for matching

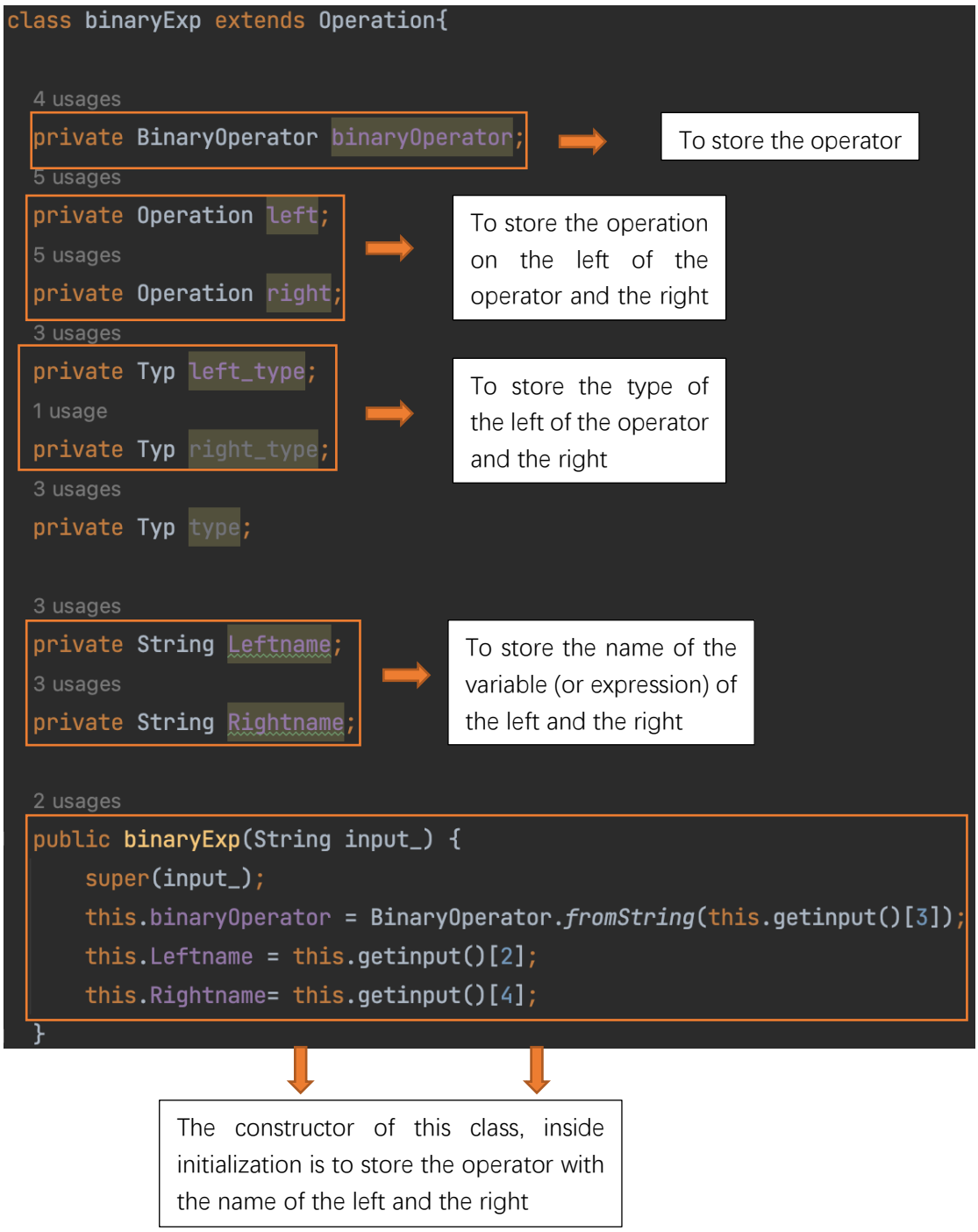
The inside calculation of each small case

```
// int to boolean
public boolean bool_calculate(Operation left, Operation right){
    boolean bool_result = true;
    switch (this){
        case GREATER:
            bool_result = left.getint_value() > right.getint_value();
            break;
        case GREATER_OR_EQUAL:
            bool_result = left.getint_value() >= right.getint_value();
            break;
        case LESS:
            bool_result = left.getint_value() < right.getint_value();
            break;
        case LESS_OR_EQUAL:
            bool_result = left.getint_value() <= right.getint_value();
            break;
        case EQUAL:
            bool_result = left.getint_value() == right.getint_value();
            break;
        case NOT_EQUAL:
            bool_result = left.getint_value() != right.getint_value();
            break;

    }

    return bool_result;
}
```

This method is to calculate the condition that one integer variable while the other is a Boolean variable (like calculating the result of the “x <= 10”).


class binaryExp extends Operation{
4 usages
private BinaryOperator binaryOperator;
To store the operator
5 usages
private Operation left;
5 usages
private Operation right;
To store the operation on the left of the operator and the right
3 usages
private Typ left_type;
1 usage
private Typ right_type;
To store the type of the left of the operator and the right
3 usages
private Typ type;
3 usages
private String Leftname;
3 usages
private String Rightname;
To store the name of the variable (or expression) of the left and the right
2 usages
public binaryExp(String input_) {
super(input_);
this.binaryOperator = BinaryOperator.fromString(this.getinput()[3]);
this.Leftname = this.getinput()[2];
this.Rightname= this.getinput()[4];
}
The constructor of this class, inside initialization is to store the operator with the name of the left and the right


```
public void printlist(ArrayList<String> list) {
  Checking.listExpression(list, this.Leftname);
  Checking.listExpression(list, this.Rightname);

  if (!list.contains(this.getInputString())) {
      list.add(this.getInputString());
  }
}
```

This “printList” method is created to work for the following function “program list”.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *expName* is valid
3. Whether the *bop* is valid
4. If the bop is int type, whether the operand follows int type *expRef* (is int string and in the range -99999~99999 or the expression name is valid)
5. If the bop is bool type, whether the operand follows bool type *expRef* (is bool string or the expression name is valid)

6. Whether the expression with the same *expName* is already stored

For execute error handling, it has the following checking:
Because the expRef of variable and expression can not be detected in the user input formatting handling, so we need to further check

1. Whether the 2 expRef are of the same type
2. Whether the operator is corresponding to the data type

For the calculation, we need to round the result to -99999 ~ 99999. Also, handling divided by 0 exception

[REQ3]

1) The requirement is implemented.
2) Implementation details.

Initialize the Enum, prepare for the recognition of the operator

Get in the input and match it with the Enum

```
public enum UnaryOperator {
    1 usage
    PLUS( symbol: "#"), MINUS( symbol: "~"),NOT( symbol: "!");
    2 usages
    private final String symbol;

    3 usages
    UnaryOperator(String symbol) { this.symbol=symbol; }
```

Use the switch to match the input with the correspondin g operation

```
public int calculateInt(Operation op) {
    int result = 0;
    switch (this) {
    case PLUS:
        result = +op.getint_value();
        break;
    case MINUS:
        result = -op.getint_value();
        break;
    }return result;
}
```

The MINUS case operation codes

The operation of NOT for the Boolean type

```
public boolean calculateBool(Operation op) {
    boolean result = true;
    switch (this) {
    case NOT:
        return !op.getbool_value();
    }return result;
}
```

The loop inside to check all the operators

Match the corresponding operator in the Enum

```
public String getSymbol() { return this.symbol; }

public String toString() { return getSymbol(); }

public static UnaryOperator fromString(String symbol){
    for(UnaryOperator operator: UnaryOperator.values()){
        if(operator.getSymbol().equals(symbol)){
            return operator;
        }
    }
    return null;
}
```

```
class unaryExp extends Operation{
    3 usages
    private UnaryOperator unaryOperator;
    4 usages
    private Operation exp;
    3 usages
    private Typ type;
    2 usages
    private String expName;
    2 usages
    public unaryExp(String input_) {
        super(input_);
        this.unaryOperator = UnaryOperator.fromString(this.getinput()[2]);
        this.expName = this.getinput()[3];
    }
```

Store the unary operator

Store the information that will be executed from the input expression

The initialization of the unary expression with the constructor.

```
public void execute() {
    this.exp = Checking.getvalue(this.getinput()[3]);
    this.type =  this.exp.gettype();
    super.set_type(type);

    if (this.type == Typ.INT) {
        super.set_intValue(unaryOperator.calculateInt(exp));
    }else {
        super.set_boolValue(unaryOperator.calculateBool(exp));
    }
}
```

Get the expression and the type

Judge whether is integer and do the operations for the integer

```
public void printlist(ArrayList<String> list) {
    Checking.listExpression(list, this.expName);

    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }
}
```

This “printList” method is created to work for the following function “program list”.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *expName* is valid
3. Whether the *uop* is valid
4. If the uop is int type, whether the operand follows int type *expRef* (is int string and in the range -99999~99999 or the expression name is valid)
5. If the uop is bool type, whether the operand follows bool type *expRef* (is bool string or the expression name is valid)
6. Whether the expression with the same *expName* is already stored

For execute error handling, it has the following checking:

Because the expRef of variable and expression cannot be detected in the user input formatting handling, so we need to further check

1. Whether the operator is corresponding to the data type

For the calculation, we need to round the result to -99999 ~ 99999.

[REQ4]

1) The requirement is implemented.

2) Implementation details.

```
public class Assign extends Operation {
    1 usage
    String label;
    4 usages
    private String varName;
    3 usages
    private String expName;
    5 usages
    private Operation exp;

    public Assign(String input_){
        super(input_);
        label = getinput()[1];
        this.varName = this.getinput()[2];
        this.expName = this.getinput()[3];
    }
```

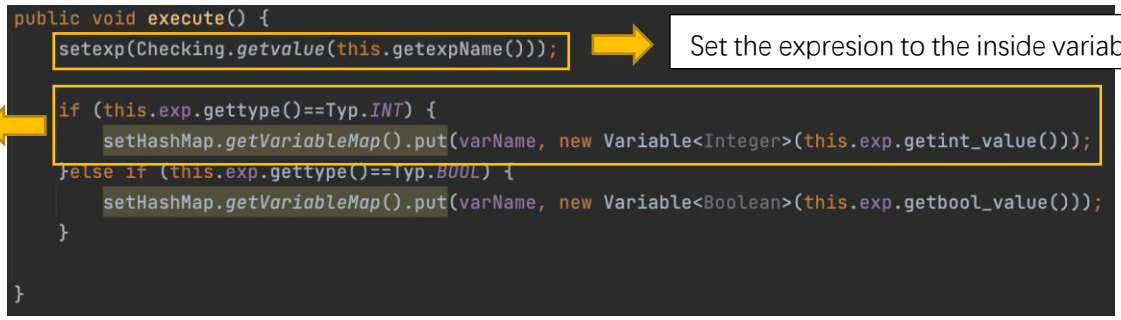
Store the label of this operation

Store the target variable

Store the matched expression

The initialization and the constructor of this function

Judge the type of the operation and use matched methods for different types of operations




```
public void printlist(ArrayList<String> list) {
    Checking.listExpression(list, this.expName);
    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }

}
```

This “printList” method is created to work for the following function “program list”.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *lab* is valid
3. Whether the *varName* is valid
4. Whether the *expRef* is valid (is int string or bool string or valid expression name)
5. Whether the statement with the same *lab* is already stored

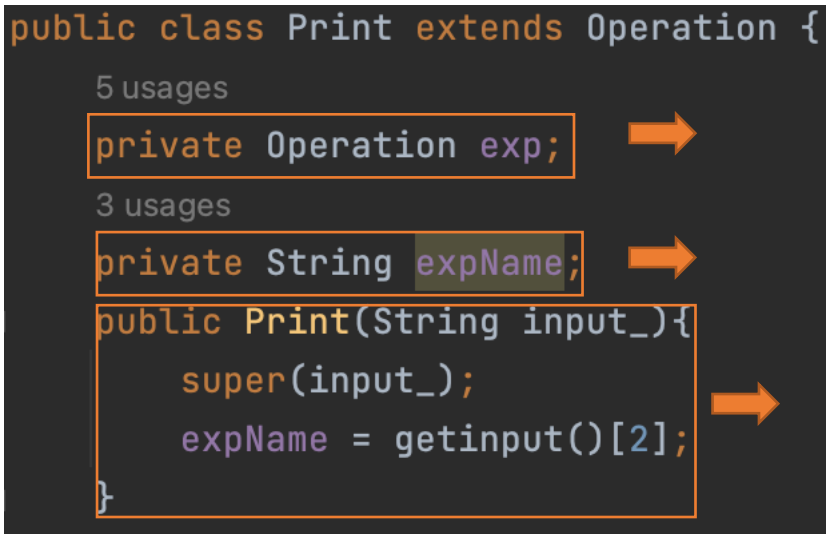
For execute error handling, it has the following checking:

1. Whether the variable has already been defined
2. Whether the data type of expression is corresponding to the variable data type

[REQ5]

1) The requirement is implemented.

2) Implementation details.



Store the expression

Store the expression name of this function

The initialization and the constructor of this function

Initialize with checking its original type

Judge the type of the operation and use matched methods for different types of operations

```
public void execute(){
    setexp(Checking.getvalue(this.getexpName()));

    if (this.exp.gettype()==Typ.INT) {

        System.out.println("["+this.exp.getint_value()+"]");
    }else if (this.exp.gettype()==Typ.BOOL) {
        System.out.println("["+this.exp.getbool_value()+"]");
    }
}
```

```
public void printlist(ArrayList<String> list) {
    Checking.listExpression(list, this.expName);

    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }
  }
```

This “printList” method is created to work for the following function “program list”.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *lab* is valid
3. Whether the *expRef* is valid (is int string or bool string or valid expression name)

4. Whether the statement with the same *lab* is already stored

For execute error handling, it has the following checking:

1. Whether the expression has already been defined (this part is done in CLASS Checking, because so many expression is needed, the Checking.getvalue will determine the value, and throw exceptions.)

[REQ6]

1) The requirement is implemented.

2) Implementation details.

```
public class Skip extends Operation{
    1 usage
    private String label;
    2 usages
    public Skip(String input_){
        super(input_);
        label = getinput()[1];
    }
```

Store the label of this operation

The initialization and the constructor of this function

```
public void execute(){

}
```

Because the function requirement for this is to skip the current operation and do nothing, so the execution of this function is nothing inside but just skip this operation.

```
public void printlist(ArrayList<String> list) {

    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }
  }
```

This “printList” method is created to work for the following function “program list”.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length

2. Whether the *lab* is valid
3. Whether the statement with the same *lab* is already stored

[REQ7]

1) The requirement is implemented.

2) Implementation details.

```
public class Block extends Operation {
    public String label;
    6 usages
    private List<String> list = new ArrayList<>();

    public Block(String input_){
        super(input_);
        label = this.getinput()[1];
        for (int i = 2,j=0; i<this.getinput().length;i++,j++){
            list.add(getinput()[i]);
        }
    }
```

Store this the label of this operation

To store every line's operation

The initialization and the constructor of this function

```
public void execute(){
    for (int i = 0; i<list.size(); i++){
        if (setHashMap.getStatementMap().containsKey(list.get(i))){
            Checking.executeStatement(list.get(i));
        }
        else {
            System.out.println("The statement doesn't exist.");
            throw new IllegalArgumentException();
        }
    }
}
```

Use loop to get every line of the block

If the statement exists, then execute matched operation

If not exist, then print the warning while catching the error

```java
public void printlist(ArrayList<String> newlist) {
    for (int i = 0; i<this.list.size(); i++){
        Checking.listStatement(newlist, this.list.get(i));
    }
    if (!newlist.contains(this.getInputString())) {
        newlist.add(this.getInputString());
    }
}
```

This “printList” method is created to work for the following function “program list”

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. For lab and every statementLab, check whether it is valid
2. Whether the statement with the same *lab* is already stored

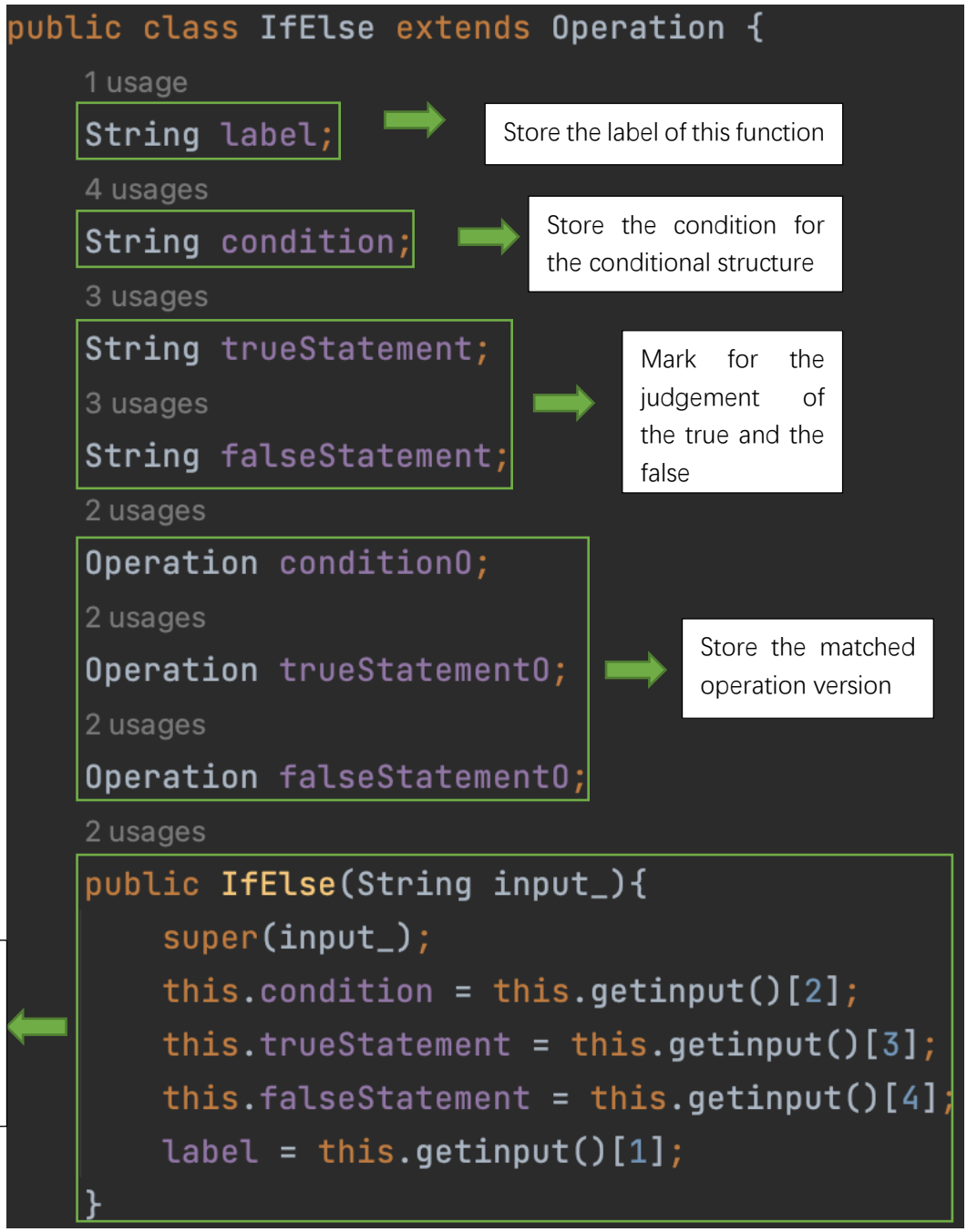
For execute error handling, it has the following checking:

1. Whether the statement in the block has already be defined (this part is done in CLASS Checking, because so many expression is needed, the Checking.executeStatement will determine the value, and throw exceptions.)

[REQ8]

1) The requirement is implemented.

2) Implementation details.


public class IfElse extends Operation {
1 usage
String label;
Store the label of this function
4 usages
String condition;
Store the condition for the conditional structure
3 usages
String trueStatement;
3 usages
String falseStatement;
Mark for the judgement of the true and the false
2 usages
Operation condition0;
2 usages
Operation trueStatement0;
2 usages
Operation falseStatement0;
Store the matched operation version
2 usages
public IfElse(String input_){
super(input_);
this.condition = this.getinput()[2];
this.trueStatement = this.getinput()[3];
this.falseStatement = this.getinput()[4];
label = this.getinput()[1];
}
The initialization and the constructor of this function

Check the judgement, if false then execute

If the condition is not Boolean type, then print the warning

```
public void execute(){

    if (Checking.getvalue(condition).getbool_value()) {
        Checking.executeStatement(trueStatement);
    }else if (!Checking.getvalue(condition).getbool_value()) {
        Checking.executeStatement(falseStatement);
    }else {
        System.out.println("Boolean types required here.");
    }
}
```

Check the judgement, if true then execute

```
public void printlist(ArrayList<String> list) {
    Checking.listExpression(list, this.condition);
    Checking.listStatement(list, this.trueStatement);
    Checking.listStatement(list, this.falseStatement);
    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }
  }
```

This “printList” method is created to work for the following function “program list”

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *lab, statementLab1, statementLab2* is valid
3. Whether the *expRef* is valid (is bool string or valid expression name)
4. Whether the statement with the same *lab* is already stored

For execute error handling, it has the following checking:

1. Because the expRef of variable and expression cannot be detected in the user input formatting handling, so we need to further check whether the expRef type is Boolean
2. Whether the statement in the block has already be defined (this part is done in CLASS Checking, because so many expression is needed, the Checking.executeStatement will determine the value, and throw exceptions.)

[REQ9]

1) The requirement is implemented.

2) Implementation details.

```
public class While extends Operation{
    private Operation statement;
    3 usages
    private String expName;
    3 usages
    private String statementName;
    2 usages
    public While(String input_) {
        super(input_);
        this.expName = getinput()[2];
        this.statementName = getinput()[3];

    }
```

Store the statement for judging whether can continue the loop

Store every line's expression

The initialization and the constructor of this function

Check the statement, if still true, then execute

```
public void execute() {
    while (Checking.getvalue(expName).getbool_value()==true) {
        Checking.executeStatement(this.getStatementName());
    }
}
```

```
public void printlist(ArrayList<String> list) {
    Checking.listExpression(list, this.expName);
    Checking.listStatement(list, this.statementName);
    if (!list.contains(this.getInputString())) {
        list.add(this.getInputString());
    }
}
```

This "printList" method is created to work for the following function "program list"

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the *lab, statementLab1* is valid
3. Whether the *expRef* is valid (is bool string or valid expression name)
4. Whether the statement with the same *lab* is already stored

For execute error handling, it has the following checking:

1. Because the expRef of variable and expression cannot be detected in the user input formatting handling, so we need to further check whether the expRef type is Boolean
2. Whether the statement in the block has already be defined (this part is done in CLASS Checking, because so many expression is needed, the Checking.executeStatement will determine the value, and throw exceptions.)

[REQ10]

1) The requirement is implemented.

2) Implementation details.

A specified class named Program was designed to realize all operations related to the program level instead of the operation level.

```
public class Program  {
//    public String label;
    10 usages
    private String program_name;
    6 usages
    private ArrayList<String> list = new ArrayList<>();
    4 usages
    public static HashSet<String> breakpoints = new HashSet<>();
//    public static HashSet<String> beforeinstruments = new HashSet<>();
//    public static HashSet<String> afterinstruments = new HashSet<>();
    public String program;
    2 usages
    private String inputString;
    7 usages
    private static Instrument instrument = new Instrument();


    public Program(String input_){
//        label = this.getinput()[1];
        this.program_name = input_.split( regex: " ")[2];
        this.program = input_.split( regex: " ")[1];
        this.inputString = input_;
    }
```

store the content of a

store all codes as strings of a program

store the label

an instance of instrument class to do code instrument

When a program is defined using the corresponding statement, all statements in this program will be stored in the array list. The program class includes the methods below, which will be specified later:

1) public void addinstrument (String lab, String pos, String expName): for instrumenting the program.

2) public void setbreakpoint (String s): for setting several breakpoints of the program.

3) public void debug (): for debugging the program
4) public void printList (): for storing all strings of the program into the array.
5) public void store (String path): for storing the program to a designated path.

After the program instance is created, it will be stored in the HashMap named programs with its label as the key and this instance as the value. It could be searched by the program label.

3) Error conditions and how they are handled.
For the user input format handling, it has the following checking:

1. String length
2. Whether the *statementLab* is valid
3. Whether the program with the same *programName* is already stored

For execute error handling, it has the following checking:

1. Whether the statement in the block has already be defined (this part is done in CLASS Checking, because so many expression is needed, the Checking.executeStatement will determine the value, and throw exceptions.)

[REQ11]
1) The requirement is implemented.
2) Implementation details.
The member method of the Program class will be executed

```
public void execute(){
    setHashMap.getVariableMap().clear();
    if (setHashMap.getStatementMap().containsKey(this.program_name)){
        this.instrument.checkBefore(program_name);
        Checking.executeStatement(program_name,instrument);
        this.instrument.checkAfter(program_name);
    }
    else {
        System.out.println("The statement doesn't exist.");
        throw new IllegalArgumentException();
    }
}
```

3) Error conditions and how they are handled.
For the user input format handling, it has the following checking:

1. String length
2. Whether the program with *programName* is already defined

For execute error handling, it has the following checking: Same as [REQ10]

[REQ12]

1) The requirement is implemented.

2) Implementation details.

```
case "list" -> ((Program) setHashMap.getProgramMap().get(inputlist[1])).printList();
```

When “list” is detected in the input string, the member method of the corresponding program instance, printList (), will be called.

```
public void printList() {
    Checking.listStatement(this.list,this.program_name);
    list.add(this.inputString);
    for (String s : list) {
        System.out.println(s);
    }
}
```

The listStatement method in the Checking class will be called. It will read the commands in this program and call the corresponding printlist (), which will add the statement string to the list. After this operation, the list contains all statements of the program, and each of them will be printed sequentially.

```
public static void listStatement(ArrayList<String> list,String sName) {
    switch (((Operation) setHashMap.getStatementMap().get(sName)).getLabel()) {
        case "vardef":
            ((varDef) setHashMap.getStatementMap().get(sName)).printlist(list);
            break;
        case "assign":
            ((Assign) setHashMap.getStatementMap().get(sName)).printlist(list);
            break;
        case "print":
            ((Print) setHashMap.getStatementMap().get(sName)).printlist(list);
            break;
        case "skip":
            ((Skip) setHashMap.getStatementMap().get(sName)).printlist(list);
            break;
        case "block":
            ((Block) setHashMap.getStatementMap().get(sName)).printlist(list);
            break;
        case "if":
            ((IfElse) setHashMap.getStatementMap().get(sName)).printlist(list);
            break;
        case "while":
            ((While) setHashMap.getStatementMap().get(sName)).printlist(list);
    }
```

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the program with *programName* is already defined

For execute error handling, it has the following checking: see Checking.listExpression and Checking.listStatement, if we can find the expressionName or StatementName, we add to the list. If not, we no nothing.

[REQ13]

1) The requirement is implemented.

2) Implementation details.

```
public void store(String path) {
    Checking.listStatement(this.list,this.program_name);

    BufferedWriter writer = null;
    File file = new File(path);
    // create a new file if the file doesn't exist
    if(!file.exists()){
        try {
            file.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    // write the data into the file
    try {
        writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file, append: false), StandardCh
        for(int i=0;i<list.size();i++) {
            writer.write(list.get(i));
            writer.newLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }finally {
        try {
            if(writer != null){
                writer.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

write the data into the file

```
case "store" -> {
    ((Program) setHashMap.getProgramMap().get(inputlist[1])).store(inputlist[2]);
```

When the word “store” is detected in the input, the program will call the member method, store (), of the corresponding program to store the program. The third word of the input is considered the target path, while the method will use the BufferedWriter method to store the program using the UTF-8 encoding method. During the execution of this method, IO exceptions will be cached if any.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the program with *programName* is already defined

For execute error handling, it has the following checking:

```java
BufferedWriter writer = null;
File file = new File(path);
// create a new file if the file doesn't exist
if(!file.exists()){
    try {
        file.createNewFile();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
// write the data into the file
try {
    writer = new BufferedWriter(new OutputStrea
    for (String s : list) {
        writer.write(s);
        writer.newLine();
    }
} catch (IOException e) {
    e.printStackTrace();
}finally {
    try {
        if(writer != null){
            writer.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
```

[REQ14]

1) The requirement is implemented.

2) Implementation details.

```java
public class LoadProgram {
    2 usages
    private final String path;
    1 usage
    private final String programName;
    1 usage
    public LoadProgram(String input_){
        String[] input = input_.split( regex: " ");
        path = input[1];
        programName = input[2];
    }

    1 usage
```

The LoadProgram class is an individual class other than a subclass of Program. When this instance is initialized, the constructor will catch the path and program label, then call the method Load ().

```java
public void Load(){
    String Path = this.path;
    BufferedReader reader = null;
    String data = "";
    try {
        FileInputStream fileInputStream = new FileInputStream(Path);
        InputStreamReader inputStreamReader = new InputStreamReader(fileInputStream, charsetName: "UTF-8");
        reader = new BufferedReader(inputStreamReader);
        String com = null;
        while ((com = reader.readLine()) != null) {
            String[] inputlist = com.split( regex: " ");
            System.out.println(com);
            switch (inputlist[0]) {
                case "vardef":
                    setHashMap.getStatementMap().put(inputlist[1], new varDef(com));
                    break;
                case "binexpr":
                    setHashMap.getExpressionMap().put(inputlist[1], new binaryExp(com));
                    break;
                case "unexpr":
                    setHashMap.getExpressionMap().put(inputlist[1], new unaryExp(com));
                    break;
                case "assign":
                    setHashMap.getStatementMap().put(inputlist[1], new Assign(com));
                    break;
                case "print":
                    setHashMap.getStatementMap().put(inputlist[1], new Print(com));
                    break;
                case "skip":
                    setHashMap.getStatementMap().put(inputlist[1], new Skip(com));
                    break;
                case "block":
                    setHashMap.getStatementMap().put(inputlist[1], new Block(com));
```

The file input stream will read the file line by line and store them sequentially into the HashSet of statements. It should store all statements, including the program define statement.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length

For execute error handling, it has the following checking:

1. Whether the file path is valid
2. All the input checking for all statements

[REQ15]

1) The requirement is implemented.

2) Implementation details.

```java
case "quit" -> System.exit( status: 0);
```

When the word “quit” is detected in the input string, the system will call System.exit(0) to terminate the current process immediately.

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length

[BON1]

1) The requirement is implemented.

2) Implementation details.

To bring out the best in JAVA, we directly add a new mode in the checking class to achieve the function of the debug, set breakpoints and the inspect function.

```
public static void debugStatement(String name, HashSet<String> set,String program,Instrument instrument) {
    System.out.println(((Operation) setHashMap.getStatementMap().get(name)).getLabel());
    if (set.contains(name)){
          while(!inputlist[0].equals("debug")){
        boolean repeat = true;
        while(repeat){
            System.out.print('>');
            Scanner scanner = new Scanner(System.in);
            String com= scanner.nextLine();
            String[] inputlist = com.split( regex: " ");
            switch (inputlist[0]){
                case "togglebreakpoint" -> InputProcessing.togglebreakpointChecking(com);
                case "inspect" -> {
                    if (inputlist.length != 3) {
                        System.out.println("Invalid command:" + com);
                    } else if (!inputlist[1].equals(program)){
                        System.out.println("Error: "+com);
                        System.out.println("Not current debugging program: "+inputlist[1]);
                    } else if (setHashMap.getVariableMap().containsKey(inputlist[2])) {
                        if (((Variable<?>) setHashMap.getVariableMap().get(inputlist[2])).getvalue() instanceof Integer) {
                            System.out.println("<"+((Variable<Integer>) setHashMap.getVariableMap().get(inputlist[2])).getvalue()+">");
                        }else{
                            System.out.println("<"+((Variable<Boolean>) setHashMap.getVariableMap().get(inputlist[2])).getvalue()+">");
                        }
                    }else {
                        System.out.println("Error: "+com);
                        System.out.println("Variable hasn't defined:"+inputlist[2]);
                    }
```

```
                }case "debug" ->{
                    if (inputlist.length != 2) {
                        System.out.println("Invalid command:" + com);
                    } else if (!inputlist[1].equals(program)){
                        System.out.println("Error: "+com);
                        System.out.println("Not current debugging program: "+inputlist[1]);
                    } else {
                        repeat = false;
                    }
                }
            }
        }
    }
    try{
        setHashMap.getStatementMap().get(name);
    }catch (NullPointerException e){
        System.out.println("This statement is not defined:"+name);
        System.exit( status: 0);
    }
```

```
try{
    setHashMap.getStatementMap().get(name);
}catch (NullPointerException e){
    System.out.println("This statement is not defined:"+name);
    System.exit( status: 0);
}
switch (((Operation) setHashMap.getStatementMap().get(name)).getLabel()) {
    case "vardef" -> ((varDef) setHashMap.getStatementMap().get(name)).debug(set, program, instrument
    case "assign" -> ((Assign) setHashMap.getStatementMap().get(name)).debug(set, program, instrument
    case "print" -> ((Print) setHashMap.getStatementMap().get(name)).debug(set, program, instrument);
    case "skip" -> ((Skip) setHashMap.getStatementMap().get(name)).debug(set, program, instrument);
    case "block" -> ((Block) setHashMap.getStatementMap().get(name)).debug(set, program, instrument);
    case "if" -> ((IfElse) setHashMap.getStatementMap().get(name)).debug(set, program, instrument);
    case "while" -> ((While) setHashMap.getStatementMap().get(name)).debug(set, program, instrument);
}
```

3) Error conditions and how they are handled.

Debug:

For the user input format handling, it has the following checking:

For the outer loop:

1. String length
2. Whether the program with *programName* is already defined

For the inner debug:

1. String length
2. Whether the program with *programName* is what we are debugging

For execute error handling, it has the following checking:

1. Same as Checking.execute, but with Checking.debugStatement

Togglebreakpoint:

For both outer and inner debug

For the user input format handling, it has the following checking:

1. String length
2. Whether the program with *programName* is already defined

Inspect:

For the user input format handling, it has the following checking:

For the inner debug:

1. String length
2. Whether the program with *programName* is what we are debugging
   Whether the variable with *varName* is already defined

[BON2]

1) The requirement is implemented.

2) Implementation details.

```java
/**
 * Instrument is class to store information for instrument
 */
public class Instrument {
    private static final ArrayList<String> instrumentStat = new ArrayList<>(); // which statement to be instrumented
    private static final ArrayList<String> instrumentVal = new ArrayList<>();  // instrument which value
    private static final ArrayList<String> instrumentPos = new ArrayList<>(); // before or after

    /**
     * constructor
     */
    Instrument(){

    }
```

The constructor of this separate class

```java
    /**
     * @param stat statement string
     * @param pos position string
     * @param val expression string
     */
    public void setInstrument(String stat, String pos, String val){

        //input all instrument, allow multiple statements for one
        instrumentStat.add(stat);
        instrumentVal.add(val);
        instrumentPos.add(pos);
    }
```

Use loop to check every line of the

```java
    /**
     * @param program_name to specify which program is corresponds to
     */
    public void checkBefore(String program_name){
        if (setHashMap.getStatementMap().containsKey(program_name)) { // if statement exists
            for (int i = 0; i < instrumentPos.size(); i++) {
                if (program_name.equals(instrumentStat.get(i))) // if statement is intrumented
                    if (instrumentPos.get(i).equals("before")) {
                        Operation exp = Checking.getvalue(instrumentVal.get(i));
                        if (exp.gettype()==Typ.INT) {

                            System.out.println("{"+exp.getint_value()+"}");
                        }else  {
                            System.out.println("{"+exp.getbool_value()+"}");
                        }
                    }
            }
        }
    }
```

```java
/**
 * @param program_name to specify which program is corresponds to
 */
public void checkAfter(String program_name){
    if (setHashMap.getStatementMap().containsKey(program_name)) {
        for (int i = 0; i < instrumentPos.size(); i++) {
            if (program_name.equals(instrumentStat.get(i)))
                if (instrumentPos.get(i).equals("after")) {
                    Operation exp = Checking.getvalue(instrumentVal.get(i));
                    if (exp.gettype()==Typ.INT) {
                        System.out.println("{"+exp.getint_value()+"}");
                    }else  {
                        System.out.println("{"+exp.getbool_value()+"}");
                    }
                }
        }
    }
}
```

Use loop to check every line of the

3) Error conditions and how they are handled.

For the user input format handling, it has the following checking:

1. String length
2. Whether the position is “after” or “before”
3. Whether the program with *programName* is already defined

For execute error handling, it has the following checking:

1. Whether the statement with *statementLab* is already defined

3 User Manual

About the Manual

Simple Language is a command-line-based interpreter, that can be defined and interpreted by Simple Language. The User Manual is as guidance for users with a complete understanding of Simple Language.

Audience

The user manual is designed for beginning learners, or for experienced programmers to quickly get a good understanding of how to use the Simple Language.

Prerequisites

The user is assumed with basic knowledge of computer programming and has already set up the Java environment with the simple language to get the command line.

Execute Simple Program Online

Try the simple sample below to print the value of x which is 10, to experience how the code will be executed in the Simple Language.

```
>vardef vardef1 int x 10
>print print1 x
>block block1 vardef1 print1
>program program1 block1
>execute program1
```

Overview

Simple Language is a command-line-based interpreter, which is for programmers to write in a simple programming language. The language supports limited types of data, expressions, and statements. Simple Language runs based on the java package, which can be expressed on platforms like Windows, Mac OS, and some other versions of UNIX.

Local Environment Set-up

After running the provided Java Simple Language package, the command line will be shown on the screen to be written, user can input ‘quit’ to stop the program.

Try Simple Language Program

Let us develop a simple language program first.
Print the following command line by line, and click 'Enter' after each line finished.

```
>vardef vardef1 int x 10
>print print1 x
>block block1 vardef1 print1
>program program1 block1
>execute program1
```

You will be able to see the value of x, which is 10 now.

```
[10]
>
```

Basic Syntax

Data Types –
Integer – keyword: int Decimal integer.
Boolean – keyword: bool true and false
Variable – define or assign a variable to save integer value or Boolean value
Expression – Saved as expression reference, to calculate the result of different variables
Statement – Saved in Statement Lab which can be executed as the higher-level order
Program – package the codes which can be executed

Identifiers
– cannot use keywords, name of integer or boolean value, like "true".

- Should begin with a letter (from a to z or from A to Z), after the beginning letter, can be combined by number or symbol.
- the length of an identifiers should be no longer than 8.
- Legal identifiers: vardef1, printeven, block1
- Illegal identifiers: true, -1.

Simple Language Keywords

Using Space to split each of the keyword.

vardef	binexpr	unexpr	assign	print	skip	block
if	while	program	execute	list	quit	debug
toggebreakpoint	inspect	instrument	int	bool		

vardef - define a value by integer or boolean value ,
e.g: vardef vardef1 int x 100
Command: *vardef label type varName expRef*

assign – assign a new variable value by variables
e.g: assign assign1 x exp2
Command: *assign lab varName expRef*

binexpr - calculating+, -, *, /, >, >=, <=, ==, !=, % that can be applied to integer value, and &&, ||, ==, and != that can be applied to boolean value.
e.g: binexpr exp1 x * 20
Command: *binexpr expName expRef1 bop expRef2*

unexpr – can process the integer value with #, ~, +, -, or process boolean value with !.
e.g: unexpr exp2 ~ exp1
Command: *unexpr expName uop expRef1*

print – print the expression value
e.g: print print1 exp2
Command: *print lab expRef*

skip – doing noting and continuing to run the program
e.g: skip skip1
Command: *skip lab*

block – save all usable commands in a block
e.g: block block1 assign1 skip1
Command: *block lab statementLab1 … statementLabn*

if – check the result, as a condition to execute the including command.
e.g: if if1 exp5 block1 print1
Command: *if lab expRef statementLab1 statementLab2*

while – define a new loop, when the loop is executed, the expression will be repeatedly evaluated.
e.g: while while1 true block1
Command: *while lab expRef statementLab1*

program – to package the command which will be executed as a whole
e.g: program program1 while1
Command: *program program1 while1*

execute – execute the content commands of the program
e.g: execute program1
Command: *execute programName*

list – list all the commands in the program
e.g: list program1
Command: *list programName*

store – storing the defined Simple Program into a file
e.g: store program1 d:\prog1.simple
Command: *store programName path*

load – Loading a defined Simple Program from a file.
e.g: load d:\prog1.simple program1
Command: *load path programName*

quit – Terminate the interpreter
Command: quit

togglerbreakpoint – set a breakpoint, then use debug module, the
e.g: togglebreakpoint program1 block1
Command: *togglebreakpoint programName statementLab*

debug – begin to execute the program in debug module
e.g: debug program1
Command: *debug programName*

inspect – can check the procedure value of each variable after the last breakpoint
e.g: inspect program1 x
Command*: inspect programName varName*

instrument – print the value before of after a statement
e.g: instrument printeven block1 after 5
Command*: instrument programName after/before varName*

Execute Simple Program

To execute a command, the block should be first used to package them, then the commands should be saved in a program like program1, and the execution can only execute the program.
The Sample command is as below:

```
>vardef vardef1 int x 10
>print print1 x
>block block1 vardef1 print1
>program program1 block1
>execute program1
```

Expression and Operators

Operator and Function

SIMPLE language can recognize 15 operators.

Each Operator has different functions, shown below.

Operator	function
+	addition
-	subtraction
*	multiplication
/	division
%	remainder
>	larger
>=	Larger or equal to
<	Less than
<=	Less or equal to
==	equal to
!=	not equal to
&&	and
||	or
#	Unary postive

~	Unary negative
!	NOT

Input and Output

Different operators can only be applied to specified data types.
The integer value can be saved in variables, or directly used.
The boolean value can be saved in expression, variable, or directly used.

Operator Type	Operator	Input data type	Output data type
Binary	+	Integer	Integer
Binary	-	integer	Integer
Binary	*	integer	Integer
Binary	/	integer	Integer
Binary	%	integer	Integer
Binary	>	integer	Boolean
Binary	>=	integer	Boolean
Binary	<	integer	Boolean
Binary	<=	integer	Boolean
Binary	==	integer	Boolean
Binary	!=	Integer	Boolean
Binary	&&	Boolean	Boolean
Binary	||	Boolean	Boolean
Unary	#	integer	integer
Unary	~	integer	integer

Unary	!	Boolean	Boolean

Sample Program

```
>vardef vardef1 int x 10
>binexpr exp1 x + 30
>print print1 exp1
>block block1 vardef1 print1
>program program1 block1
>execute program1
```

The execution result will be:

```
[40]
>
```

List

'List' function can only call the program name, and the command stored in the matched program and block will all be listed out.

```
>vardef vardef1 int x 0
>binexpr exp1 x % 2
>binexpr exp2 exp1 == 0
>print print1 x
>skip skip1
>if if1 exp2 print1 skip1
>binexpr exp3 x + 1
>assign assign1 x exp3
>block block1 if1 assign1
>binexpr exp4 x <= 10
>while while1 exp4 block1
>block block2 vardef1 while1
>program printeven block2
>list printeven
```

the execution result will be:

```
vardef vardef1 int x 0
binexpr exp4 x <= 10
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
while while1 exp4 block1
block block2 vardef1 while1
program printeven block2
>
```

If and While Loop

Ifelse:

When being executed, the following commands when the condition is true, else execute the other commands.

while: when the condition is satisfied, the inner command will be repeated executed until the condition is not true.

Sample Program

```
>vardef vardef1 int x 0
>binexpr exp1 x % 2
>binexpr exp2 exp1 == 0
>print print1 x
>skip skip1
>if if1 exp2 print1 skip1
>binexpr exp3 x + 1
>assign assign1 x exp3
>block block1 if1 assign1
>binexpr exp4 x <= 10
>while while1 exp4 block1
>block block2 vardef1 while1
>program printeven block2
>execute printeven
```

The execution Process:

1. x = 0
2. while: Condition: x <= 10; Repeat: Block1
3. Block1:
   If: Condition: x%2 == 0, execute: print x; else continue execute.
   Then make x = x + 1
4. Repeat for 10 rounds, and print only when x is the multiple of 2.

The execution Result will be:

```
[0]
[2]
[4]
[6]
[8]
[10]
>
```

Debug and Instrument

Debug

The debugger has three functions: togglebreakpoint, debug, and inspect. The debug should be used after the breakpoint, and each time when debugging a program, the execution will stop at the breakpoint, once debugging again, the execution will be stopped at the next breakpoint. The 'inspect' is for checking the wondering value at the current breaking moment. Again, if the same breakpoint command is executed twice at the same breaking place, the breakpoint will be removed, the command is as following:

```
>vardef vardef1 int x 0
>binexpr exp1 x % 2
>binexpr exp2 exp1 == 0
>print print1 x
>skip skip1
>if if1 exp2 print1 skip1
>binexpr exp3 x + 1
>assign assign1 x exp3
>block block1 if1 assign1
>binexpr exp4 x <= 10
>while while1 exp4 block1
>block block2 vardef1 while1
>program printeven block2
>togglebreakpoint printeven if1
>debug printeven
>inspect printeven x
>debug printeven
>inspect printeven x
>togglebreakpoint printeven if1
>debug printeven
```

The execution result will be:

```
>inspect printeven x
<0>
>debug printeven
[0]
>inspect printeven x
<1>
>togglebreakpoint printeven if1
>debug printeven
[2]
[4]
[6]
[8]
[10]
>
```

Instrument

print out a value before or after the statement from StatementLab is executed. Sample code is shown as below:

```
>vardef vardef1 int x 0
>binexpr exp1 x % 2
>binexpr exp2 exp1 == 0
>print print1 x
>skip skip1
>if if1 exp2 print1 skip1
>binexpr exp3 x + 1
>assign assign1 x exp3
>block block1 if1 assign1
>binexpr exp4 x <= 10
>while while1 exp4 block1
>block block2 vardef1 while1
>program printeven block2
>instrument printeven block1 after 5
>execute printeven
```

and the result is as following:

```
[0]
{5}
{5}
[2]
{5}
{5}
[4]
{5}
{5}
[6]
{5}
{5}
[8]
{5}
{5}
[10]
{5}
>
```

Store and Load

Load means to download information from the existing local disk, and store means to store in a an assigned place.
The command shown as below: with the keyword store and load, and the saving disk.

```
>load d:\prog1.simple printeven
>store printeven d:\prog2.simple
```

Execution Result:
The downloading will be saved in the referred document.